1.0 Introduction

Computer vision is a very important problem in modern society. It has applications in many disciplines, from analyzing targets found in satellite images, to inspecting circuit boards to determine their quality, to analyzing photos on Facebook in order to guess who the faces in them belong to.

We wanted to gain a better understanding of image recognition, but as there are already so many guides and solutions in existence, we decided we would learn about the subject by applying it to another real-world problem.

In modern robotics, it is very difficult to write a program that can autonomously seek out a target and find its own path to its goal. There exist many sensors that can give a robot information about the environment surrounding it, such as encoders, gyroscopes, accelerometers, and ultrasonic rangefinders. However, even with all these sensors, all that most robots can do is follow a pre-programmed path, and use their sensors to avoid obstacles.

To this end, our goal was to solve a simple problem in the domain of computer vision, the detection of balls of a certain color. Then, we wanted to implement our solution in hardware, and make a robot with a camera that could autonomously track a ball, turn towards it, and drive up to it.

2:Hough Transform

2. Software Implementation of the Hough Transform

2.1 Approach

We first detect edges using the Sobel operator [1], and then we apply the Hough circle transform to detect circles in the image [2.] The following steps we took will be covered in more detail under section 2.2.

1. Blurring image with the Gaussian kernel
2. Using the Sobel operator to find edge points
3. Applying Hough circle transform
4. Processing the results of the transform

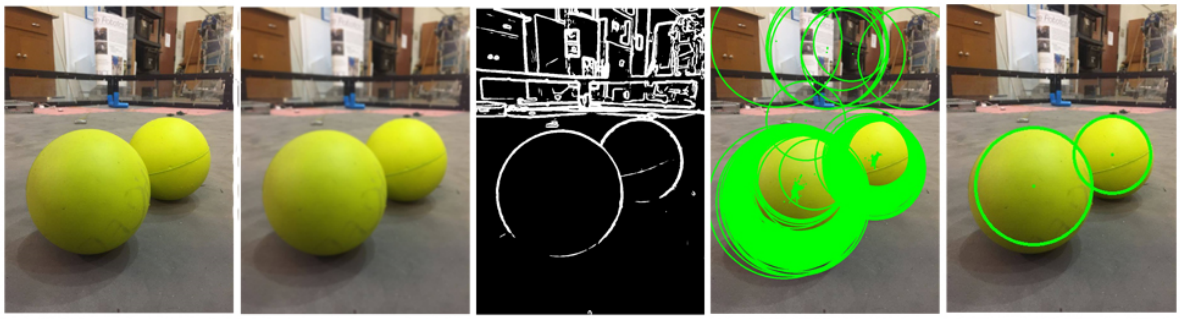


Fig 2.1 From left to right: Original image, blurred image, image Edges, Hough transform results, processing the results of the Hough transform.

2.2 Software Implementation

Software implementation was done with Python. The following Python libraries were used.

- Numpy version 1.10.1
- Scipy version 0.16.0
- OpenCV version 3.0.0

2.2.1 Blurring the image with the Gaussian kernel

Blurring the image removes noise and filters out background details. To do this, we blur the image with a Gaussian kernel.

0.002969	0.013306	0.021938	0.013306	0.002969
0.013306	0.059634	0.09832	0.059634	0.013306
0.021938	0.09832	0.162103	0.09832	0.021938
0.013306	0.059634	0.09832	0.059634	0.013306
0.002969	0.013306	0.021938	0.013306	0.002969

Fig 2.2 5x5 Gaussian kernel with a standard deviation of 1

Code

```
def blur(img, blursize):
    """
    :param img: image to be blurred
    :param blursize: size of gaussian kernel to blur with
    :return: blurred image
    """
    # ============================
    # Generate Gaussian kernel
    # ============================
    gauss = cv2.getGaussianKernel(blursize, blursize/3)
    gauss = np.outer(gauss, gauss)
    # ============================
    # Filter with Gaussian kernel
    # ============================
    img = cv2.filter2D(img, -1, gauss)
    return img
```

2.2.2 Using the Sobel operator to find edge points

Edges occur at large changes in pixel value, so by finding local maximums in the image gradient's magnitude, we can detect edge points. This concept is illustrated in 1 dimension (1D) in Fig 2.3.

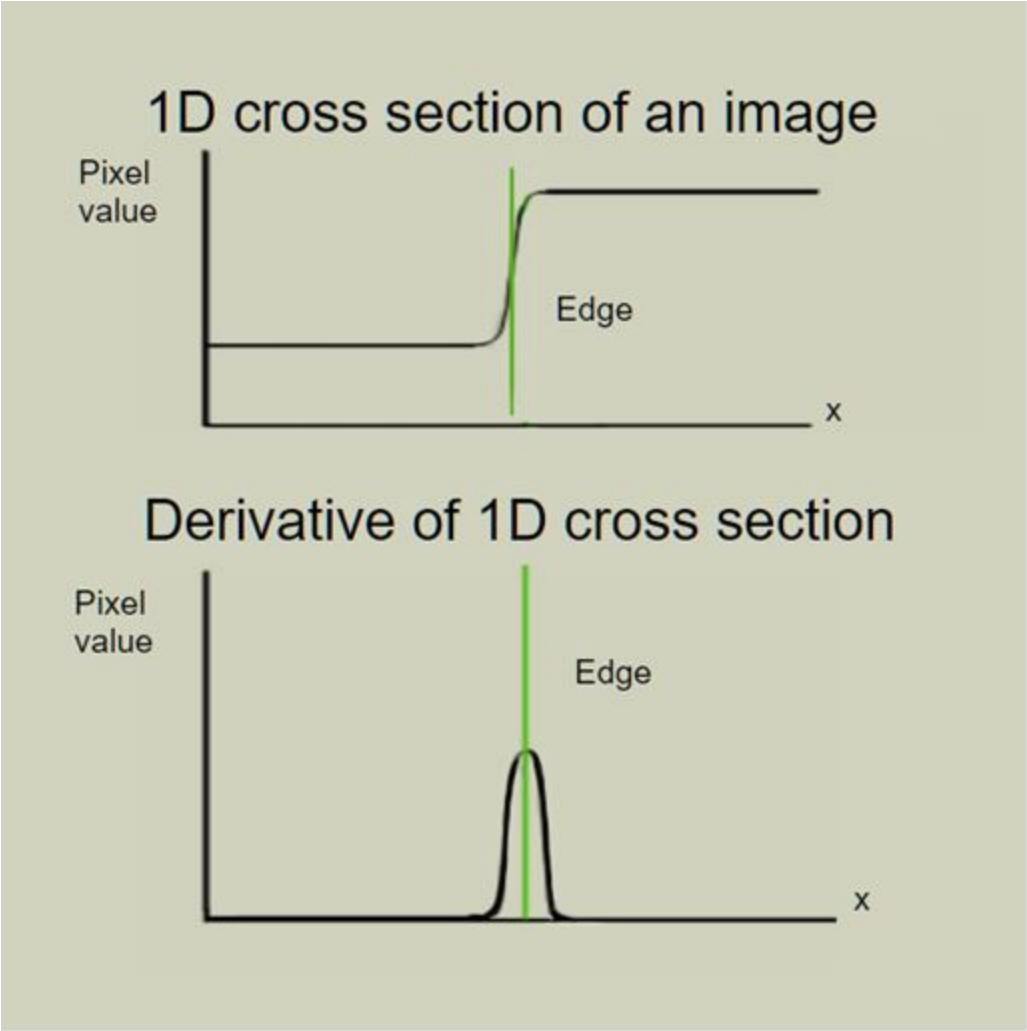


Fig 2.3 Edges occur at large changes in pixel value, or, local maximums in the derivative.

Convolving an image with the Sobel operator gives us the approximate partial derivatives in the x and y directions. Let us define The Sobel operator as Sx for the x direction and Sy for the y direction, let our define our image as A, and let us define Bx and By as the partial x and partial y, respectively, derivative approximations of A.

Sx =

-1	0	1
-2	0	2
-1	0	1

Sy =

-1	-2	-1
0	0	0
1	2	1

Fig 2.4 Sobel operators for approximating partial x and partial y derivatives

$$Bx = Sx \star A \qquad (2.1a)$$
$$By = Sy \star A \qquad (2.1b)$$

We can find the magnitude of gradient with equation 2.2, where the square and square root are element-wise operations.

$$|B| = \sqrt{Bx^2 + By^2} \qquad (2.2)$$

Now let E be a boolean matrix of the same dimensions as the image where if a pixel is an edge, then the boolean value for the pixel is True. We calculate this matrix by determining a threshold tmag and declaring every pixel whose gradient magnitude is at least tmag an edge pixel

$$E = |B| \geq t_{mag} \qquad (2.3)$$

In our actual approach, we split the image into red, blue, and green channels. We perform edge detection on all 3 and recombine them with equation 2.4. This method allows us to find cleaner edges.

$$E = E_{blue} || E_{green} || E_{red} \qquad (2.4)$$

We can find the gradient of B with equation 2.5. Though it doesn't factor into the method of edge detection used here, finding the gradient at edges is important for reducing computational complexity for the Hough transform.

$$\Theta_E = atan2(By, Bx) \qquad (2.5)$$

Code

```
def detect_edge(img, thresh, blursize):
    """
    Detects edges in an image
    :param img: image to detect edges in
    :param thresh: minimum magnitude of gradient before it's
considered an edge
    :param blursize: size of Gaussian blur filter
    :return: boolean matrix of edges in image
    """
    # ============================
    # "Generate Sobel kernel"
    # ============================

    img = blur(img, blursize)
    sobelx = np.array([[-1, 0, 1],
              [-2, 0, 2],
              [-1, 0, 1]])

    sobely = np.array([[-1, -2, -1],
              [0, 0, 0],
              [1, 2, 1]])

    # find partial derivatives of image by convolving with
Sobel kernels
    gradImgx = np.copy(img)
    gradImgy = np.copy(img)
    gradImgy = signal.convolve2d(gradImgy, sobely, mode =
"same")/4.0
    gradImgx = signal.convolve2d(gradImgx, sobelx, mode =
"same")/4.0

    # ============================
    # "Calculate gradient angle and magnitude at each pixel"
    # ============================
    angleGrad = np.arctan2(gradImgy, gradImgx)
    magGrad = np.sqrt(np.square(gradImgy) +
np.square(gradImgx))

    isEdge = np.greater_equal(magGrad, thresh)
    return isEdge, angleGrad
```

2.2.3 Applying the Hough circle transform

A circle can be represented as a center and a radius, and the Hough circle transform transforms a image from (x, y) points to (radius, center) as shown in equation 2.6 and 2.7 and figure (Fig #).

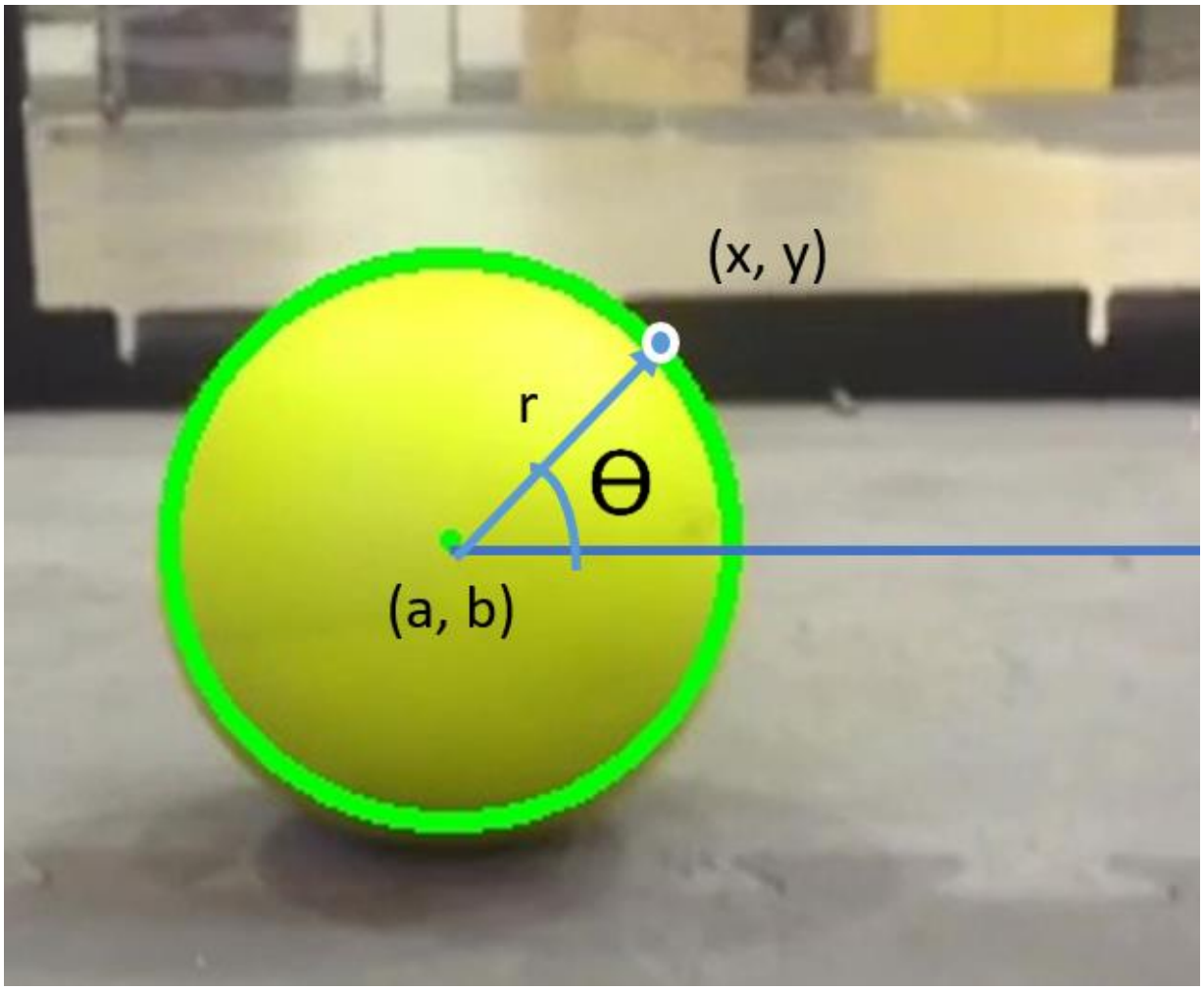


Fig 2.5 Visual of the Hough transform

$$x - rcos(\theta) = a \qquad (2.6)$$
$$y - rsin(\theta) = b \qquad (2.7)$$

Since the center and radius are both unknowns, the Hough transform tests r = from a minimum radius radMin to maximum radius radMax.

1. In an image with imgHeight rows and imgWidth columns, let V be an accumulator matrix of size (radMax, imgHeight, imgWidth).

2. For each edge point (x, y) and each possible radius r, we use equation 2.6 to calculate the center.
3. Give the resulting V(r, a, b) a vote.
4. find the local maximums in V to determine circle locations.

```
def hough_circle(colorImg, img, isEdge, angleGrad, radMin, radMax, distMin, votesMin):
    """
    Uses the Hough circle transform to detect circles in image

    :param img: Image to be operated on. size rows x cols. cv2 image format
    :param isEdge: rows x col size boolean matrix. True if pixel is edge pixel.
    :param angleGrad: rows x col size float matrix. gradient direction at each pixel.
    :param radMin: int, minimum radius of circles detected
    :param radMax: int, maximum radius of circles detected
    :param distMin: int, minimum distance between circles detected
    :param votesMin: int, minimum threshold for accumulator value. The higher this value is,
                     the less false circles detected

    :return: numpy array of circles detected. Stored as a = [[radii], [rows], [cols]]
             where
             a[0][i] is the radius of the ith circle
             a[1][i] is the row where the center of the ith circle is located
             a[2][i] is the col where the center of the ith circle is located
```

```
    """
    rows, cols = img.shape
    # Initialize accumulator matrix
    param_votes = np.zeros((radMax, rows, cols),
dtype = int)
    # Initialize accumulator matrix of local
maximums
    param_maxes = np.zeros((radMax, rows, cols),
dtype = int)

    # Perform Hough transform on each edge point
with radii ranging from radMin to radMax
    for r in range(0, rows):
        for c in range(0, cols):
            # Get gradient angle at pixel (r,c)
            # (r, c) is the proposed center for
the circle
            theta = angleGrad[r][c]
            if isEdge[r][c]:
                for rad in range(radMin, radMax):
                    # For each proposed, center,
                    # Test each possible radius
and give a vote to the resulting (rad, center)
coordinates
                    xstar = round(c - rad *
math.cos(theta))
                    ystar = round(r - rad *
math.sin(theta))
                    if xstar >=0 and xstar < cols
and ystar >=0 and ystar < rows:
                        param_votes[rad][ystar]
[xstar] += 1

                    # Add pi to the gradient angle
so the angle also points in the other direction
because
                    # who knows is the gradient is
```

```
pointing towards or away from the center of the
circle.
                    theta = theta + math.pi
                    xstar = round(c - rad *
math.cos(theta))
                    ystar = round(r - rad *
math.sin(theta))

                    # Add a vote to (rad, center)
if the center is within the bounds of the image
                    if xstar >=0 and xstar < cols
and ystar >=0 and ystar < rows:
                        param_votes[rad][ystar]
[xstar] += 1

    # Find local maximums in the accumulator
matrix.
    param_votes[param_votes < votesMin] = 0
    param_max = filter.maximum_filter(param_votes,
size = (distMin, distMin, distMin))
    param_max = np.logical_and(param_votes,
param_max)

    # Find indices of maximums
    circle_indices = np.where(param_max)

    # Will most likely find too many circles, so
use helper function to average clusters of circles
    # Also removes false positives

    circle_indices =
reduce_circles(circle_indices, distMin,
param_votes)

    total_circles = len(circle_indices[0])

    # draw detected circles onto img
```

```
    for i in range(0, total_circles):
        rad = circle_indices[0][i]
        r = circle_indices[1][i]
        c = circle_indices[2][i]
        cv2.circle(colorImg, (c, r), rad,
(0,255,0), thickness = 3)
        cv2.circle(colorImg, (c, r), 1, (0, 255,
0), thickness = 2)

    return colorImg, circle_indices
```

2.2.4 Processing the results of the transform

Often, multiple (r, a, b) within a cluster will be local maximums, and often, we'll also detect false positives. We do some processing to average clusters and remove false positives.

1. Find clusters of circles, where a cluster is defined as circles with centers a within distance of l away from each other.
2. Average the center and radius of circles in a cluster to create a new circle. Each circle in a cluster contributes its own center and radius weighted by the number of votes it received squared.
3. Remove any clusters with less than a threshold number of circles

Code

```
def reduce_circles(circle_indices, distMin, param_votes):
    """
    Helper function for hough_circle
    Averages clusters of circles

    :param circle_indices: numpy array of circles detected.
Stored as a = [[radii], [rows], [cols]]
            where
            a[0][i] is the radius of the ith circle
            a[1][i] is the row where the center of the ith
circle is located
            a[2][i] is the col where the center of the ith
circle is located
    :param distMin: int, minimum distance between circles
detected. (The function averages circles clustered within
                    a distMin x distMin square)
```

```
    :param votesMin: int, minimum threshold for accumulator
value. The higher this value is,
                     the less false circles detected
    :return: numpy array of circles
    """

    rad_indices = circle_indices[0]
    r_indices = circle_indices[1]
    c_indices = circle_indices[2]
    print rad_indices
    print r_indices
    print c_indices

    total_circles = len(rad_indices)
    print "total circles ", total_circles
    new_circle_indices = [[],[],[]]


    radtemp = 0                 # accumulator value for radii
    rtemp = 0                   # accumulator value for rows
    ctemp = 0                   # accumulator value for columns
    weightcount = 0             # accumulator value for weight
of each circle.
                                # A proposed circle with n
votes contributes n^2 weight
    count = 0                   # total number of circles
detected in a cluster.

    minThresh = total_circles / 5   # Each cluster has a count
that records number of circles in that cluster.
                                    # If count is below
minThresh, the cluster is disregarded as a false positive.

    # Remove an (radius, center) once it's been determined part
of a cluster. Iterate until no more proposed circles to
    # iterate through. Iterate backwards so deleting things
doesn't mess up indexing.
    while np.size(rad_indices)!= 0:
        # get center
        r0 = r_indices[0]
        c0 = c_indices[0]
        for i in reversed(range(0, total_circles)):
            # If iterate through all other proposed circles,
and find ones with center within a
            # distMin x distMin square.
            if abs(r0 - r_indices[i]) < distMin and abs(c0 -
c_indices[i] < distMin):
                # (radius, center) values of proposed circle
```

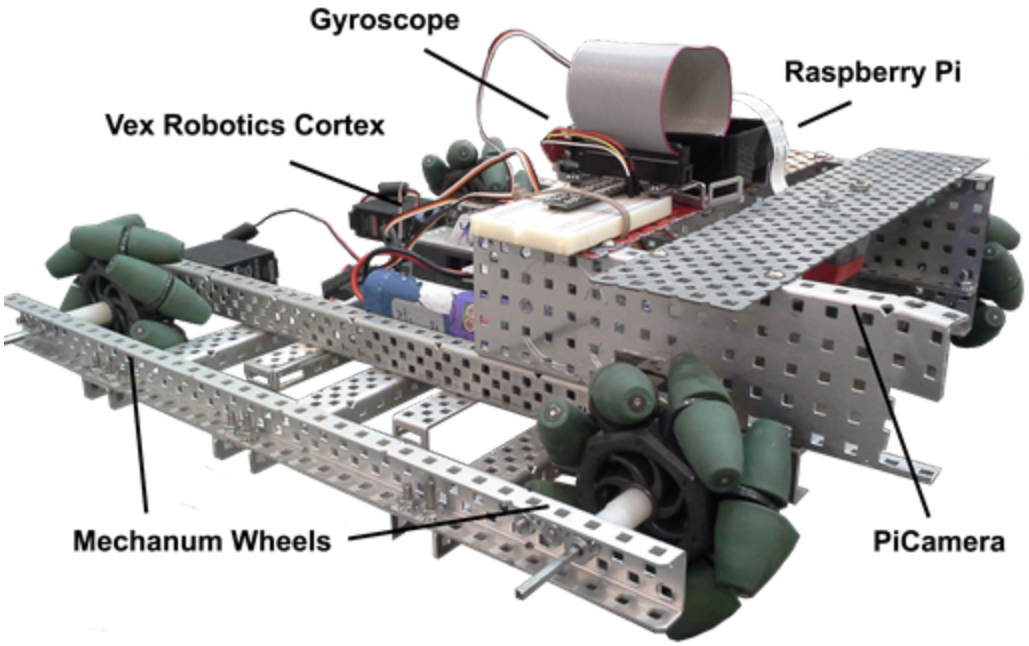
```
wwe've iterated to
                rad = rad_indices[i]
                r = r_indices[i]
                c = c_indices[i]

                # increase accumulator values, weighted by
weight
                weight = param_votes[rad][r]
[c]*param_votes[rad][r][c]
                radtemp += rad * weight
                rtemp += r * weight
                ctemp += c * weight

                # delete proposed circle from rad_indices
                rad_indices = np.delete(rad_indices, i)
                r_indices = np.delete(r_indices, i)
                c_indices = np.delete(c_indices, i)
                weightcount += weight
                count += 1
        total_circles = len(rad_indices)

        # if more circles in cluster than minThresh, average
the radii and center values to combine the cluster into
        # one circle
        if count >= minThresh:
            new_circle_indices[0].append(radtemp / weightcount)
            new_circle_indices[1].append(rtemp / weightcount)
            new_circle_indices[2].append(ctemp /weightcount)

        # Optional code. If you know you won't have circles
within circles, uncomment this code.
        # This removes circles with centers inside of other
circles.

        # for i in reversed(range(0, total_circles)):
        #     if abs(rtemp / weightcount - r_indices[i]) <
radtemp / weightcount and abs(ctemp /weightcount -
c_indices[i]) < radtemp / weightcount:
        #         rad_indices = np.delete(rad_indices, i)
        #         r_indices = np.delete(r_indices, i)
        #         c_indices = np.delete(c_indices, i)
        #     total_circles = len(rad_indices)

        # reset accumulators to 0 for the next cluster
        radtemp = 0
        rtemp = 0
        ctemp = 0
        count = 0
```

```
    weightcount = 0
return np.array(new_circle_indices)
```

3:Hardware Implementation
Usage of the Hough Circle Transform on a robot to detect yellow balls.

3.0 Hardware Introduction

This project had a two part goal: 1. Understand the Hough Transform and be able to detect circles using from-scratch code and 2. Implement the Hough Transform on hardware to create a robot that can detect yellow balls. This section will explain that hardware implementation in detail.
The Robot

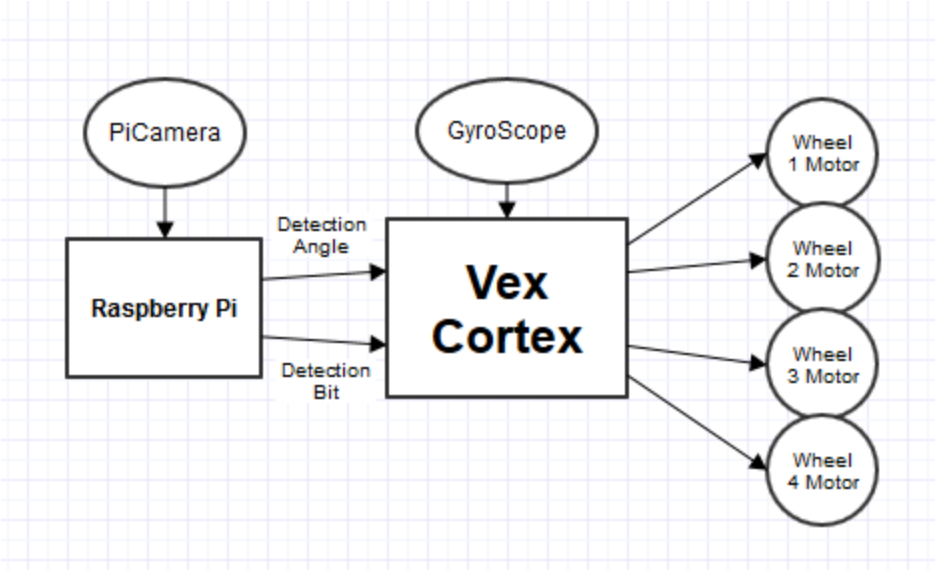


An image of our robot with important system parts labeled.

The hardware that we used for this project was largely provided by the Rice Robotics Club which uses tools from Vex Robotics. The robot four-wheeled using Mechanum wheels which allow for strafing if they are rotated in the proper directions. To drive the wheels, the Vex Cortex onboard computer

runs code in C to drive each of the four motors at the proper power levels. The cortex also receives information from a Gyroscope in order to determine what angle the robot is facing. Finally, a Raspberry Pi runs code in Python with a PiCamera to run the ball detection algorithm which transmits data to the Cortex using the UART protocol. Below we can see the block diagram for the robot's system.

Overall Hardware System



The system in its entirety. The Raspberry Pi system and Vex Cortex system are explained in detail in further sections.

3.1 Raspberry Pi System

We will now dive into the inner workings of the Raspberry Pi ball detection algorithm. The overall objective of the Raspberry Pi is to use images taken from the PiCamera and determine an angle for the robot to turn to as well as a single bit determining whether or not a ball is seen.

The Raspberry Pi System

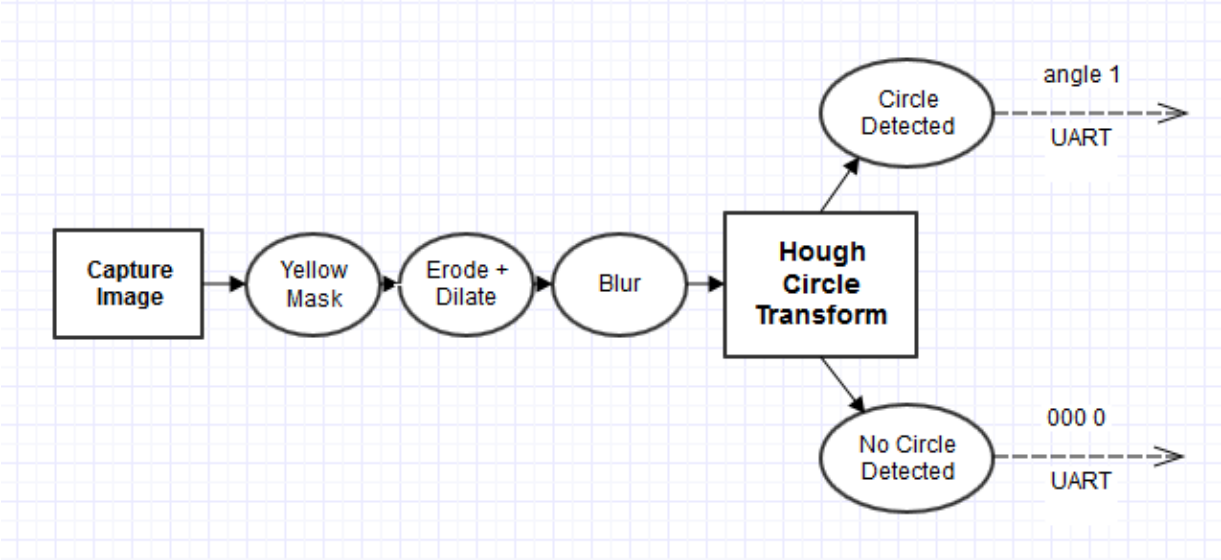


Image capture comes in through the PiCamera hardware. The output is always a 5 character string over UART to the Cortex.

The first thing the pi does is capture an image using the PiCamera. This image capture can only be done as fast as the code can run. At 360x240 pixels, the Pi can capture images at roughly 5 frames per second. Increasing that resolution decreases the frames per second and also decreases the speed that it can detect the ball. Howevever, increasing the resolution also increases the size of the ball that can be detected. A greater resolution means that the Raspberry Pi algorithm can detect balls that are either smaller or further away. At the resolution we chose (360x240) the pi can reliably detect a ball that is 2 or 3 feet away.

Once the image is captured, we need to filter out for the color yellow. This is done quite simply in the HSV (hue/saturation/value) colorspace. In good lighting, the color yellow falls in the Hue values of ~30-60, the Saturation values of ~80-255, and the Value values of ~80-255. More precise values for yellow detection can be used for more accurate results, but they depend heavily on lighting conditions on the room. For this reason, the program needs to be re-calibrated every time it enters new lighting condition. In the future we could have an auto-calibration sequence for the robot to

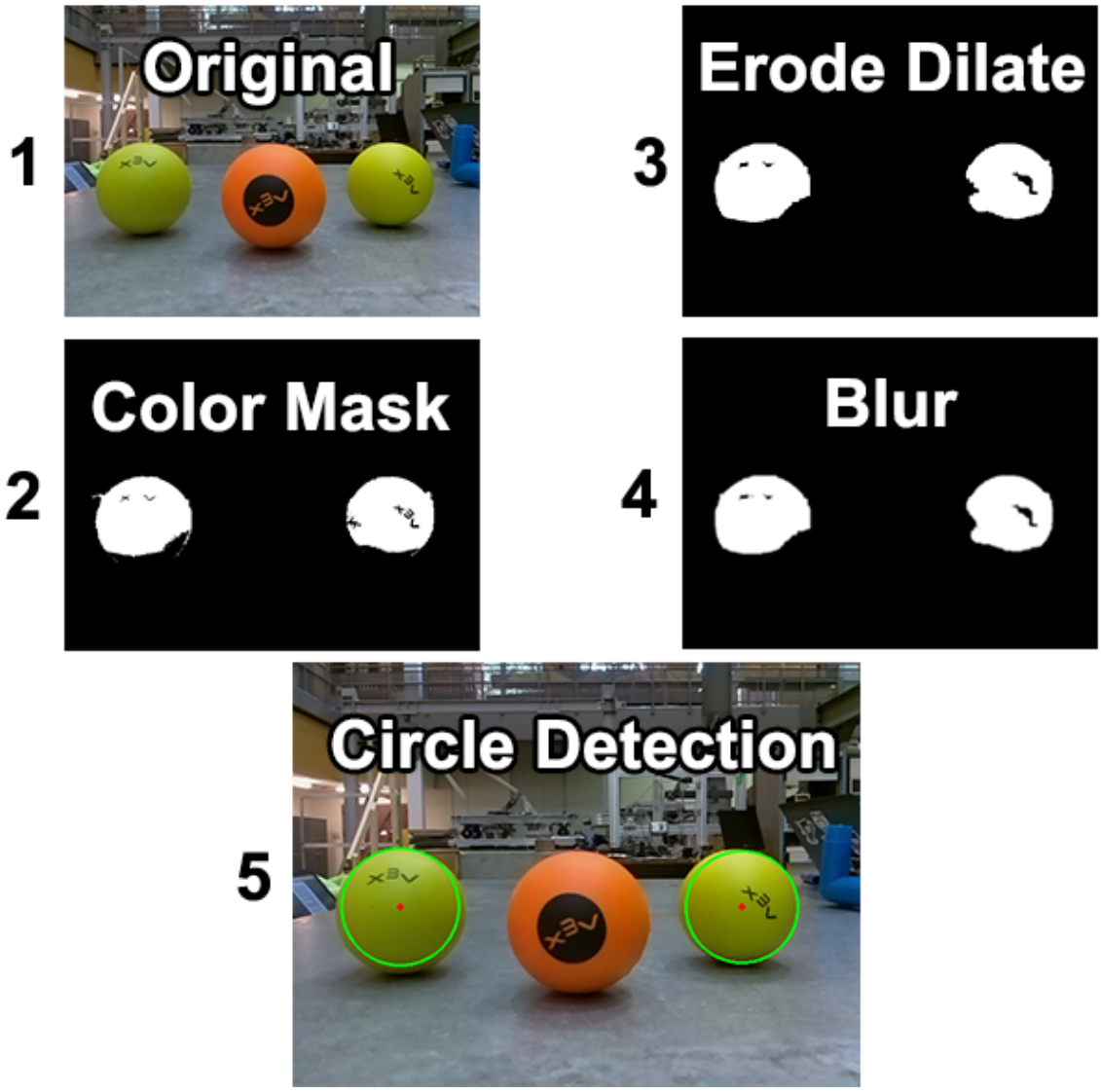
determine these values, or simply preset modes for different lighting methods.

After masking for the yellow balls, we need to filter out any noise so that the circle detector doesn't detect any false balls. We first erode, then dilate the image to get rid of most extraneous noise. This is followed by a simple Gaussian blur to help the circle detector even further.

Circle detection is done using OpenCV. We could have used the circle detector that we wrote ourselves, but it runs much slower than the one provided in OpenCV. The function

```
cv2.HoughCircles(dp,minDist,param1,param2,minRadius,maxRadius)
```

has 6 parameters that we can adjust. The dp argument corresponds to the resolution of the accumulator threshold as a simple inverse ratio between the image resolution and the accumulator resolution. The minDist argument is the minimum distance allowed between two detected circles. param1 is the threshold to be used for the Canny edge detector which is built into the function. param2 corresponds to the accumulator threshold. The smaller param2 is, the more false circles may be detected. Finally minRadius and maxRadius are the smallest and largest balls that are allowed to be detected.
Yellow Ball Detection



These are 5 images of what the Raspberry Pi can see. Each image follows a different step of the color masking and ball detection process.

Once circles are detected, the system must output an angle over UART. In this instance, there are two cases:

1. The first case is where no circles are detected. In this case, the system will output and angle of 0 and a 0 bit for circle detection. However,

occasionally a circle will miss detection for one or two frames. For this reason, we have implemented a counter that will only say that no ball was detected if the ball isn't detected for at least 4 frames.
2. The second case is where at least one ball is detected. In this case, the program chooses the largest detected yellow ball and calculates and angle to send over UART once per frame.

Once data is sent over UART, the next image is captured on the Raspberry Pi. At our chosen resolution, we can send data 5 times per second. It is then up to the Vex Cortex to drive the motors.

3.2 Robot Behavior

Finally, we will take a look at the robot itself, and how it uses the UART input in order to track and approach the ball.

The robot's vision function is a state machine with 3 states, as described below. Note that State 2 is managed by a PID control structure, using the gyroscope's current value as the input, and a setpoint of the gyroscope's value plus the angle delivered from the Raspberry Pi.

- **State 1**If the Raspberry Pi reports that there is no ball in sight, the robot will not move.
- **State 2**If a ball is detected, but the robot is not within 2 degrees of facing it, the robot will rotate to face the ball.
- **State 3**If the robot is facing the ball within tolerance, it will drive forward towards the ball.

```
void vision() {
        gyroReset(gyro->g);
        gyroPid->running = 1;
        int targetAngle = gyro->value;
//UART angle goeth here.
        int seesBall = 0;
//UART ball in sight
        int turnPow = 20;
        int drivePow = 30;
```

```
        for(ever) {
                if(joystickGetDigital(1, 8,
JOY_UP)) operatorControl();
                char* sInput = fgets(uartIn, 6,
uart1);
                if(sInput) {
                        //Decode UART input
                        char* sAngle =
strtok(sInput, " ");
                        char* sBalls =
strtok(NULL, " ");
                        seesBall = atoi(sBalls);
                        if(seesBall)
                                targetAngle =
gyro->value + atoi(sAngle);
                }
                gyroPid->setPoint = targetAngle;
                turnPow = seesBall ? gyroPid-
>output : 0;
                printf("Gyro: %d/%d, atSetpoint:
%d, output: %d\n\r", gyro->value, gyroPid-
>setPoint, gyroPid->atSetpoint, gyroPid->output);
                if(!gyroPid->atSetpoint) {
                        MOTDTFrontLeft->out = -
turnPow;
                        MOTDTFrontRight->out =
turnPow;
                        MOTDTBackLeft->out = -
turnPow;
                        MOTDTBackRight->out =
turnPow;
                }
                else if(seesBall){
                        MOTDTFrontLeft->out =
drivePow;
                        MOTDTFrontRight->out =
drivePow;
```

```
                        MOTDTBackLeft->out =
drivePow;
                        MOTDTBackRight->out =
drivePow;
                }
                if(!seesBall) {
                        targetAngle = gyro->value;
                }
                delay(20);
        }
}
```

This code uses a custom library written by the Rice Robotics Club in order to handle output to the motors, and the PID control. The source for this library can be found in **Section 4.2 Additional Resources**

4:Conclusions and Future Steps

4.0 Conclusions

As far as the Hough Transform goes, our findings were as follows:

- The success of the algorithm depends on edge magnitude threshold, Hough transform threshold, and radius parameters. and image noise. Changing thresholds results in tradeoffs between false positives and false negatives.
- For images with single balls, the algorithm can detect the ball with 95.6 % accuracy. Of the wrongly detected images, 66.7% had high amounts of motion blur.
- The algorithm can also detect balls partially cut out of the image as long as more than 60% of the ball is within the image.
- In images where multiple balls are close or overlapping, the algorithm may mistake the two balls as a single object or miss one of the balls.

The video linked in the Additional Resources section also demonstrates our Hough transform implementation in action.

4.1 Future Steps

The next primary addition to our robot is object permanence, giving it the capability to remember balls that have moved out of its field of view and rotate itself to find them again. Other future goals include recognizing when an obstacle is in front of the ball, fine-tuning the PID control in order to speed up the angling and approach to a ball, and being able to prioritize different balls based on properties like color.

4.2 Additional Resources

A video of our robot in action can be found here on Youtube.

Our poster presentation can be downloaded from this link.

Python code for the Hough transform can be found here.

The custom library used by the robot can be found here.

4.3 References

- M.K. Vairalkar and S.U. Nimbhorkar. Edge detection of images using Sobel operator. International Journal of Emerging Technology and Advanced Engineering, 2(1):291–293, 2012.
- R.O. Duda and P.E. Hart. Use of the Hough transform to detect lines and curves in pictures. Community Assoc. comput. Mach. 15 (1975), pp. 11-15.